OpenShift Broker, Proxy & Webhook

AWS IAM Role Assignment to Pods

This document provides configuration guidance and steps on how to deliver IAM role credentials to Pods running on OpenShift 4.2 & 4.3.

Architecture

Ultimately the desire to use temporary credentials stems from best practices and many other services in AWS already do this. Because of this, it is possible to use AWS services to generate or acquire temporary credentials using their own temporary credentials. This is the idea behind the lambda broker. The other components (mutating webhook & proxy sidecar) are the glue to facilitate calls to AWS Lambda.

Lambda Broker

It will use its own role and temporary credentials to generate & serve credentials to requestors. It will do this following the below rules:

1. Request is made and an Authorization header is passed in as well.
2. The Authorization header is checked against a registry (DynamoDB table) indicating the role for which credentials should be created.
3. Credentials are served back with a cache-control header set to expire 5 minutes prior to the temporary credential expiration.

Sidecar Proxy

To leverage the STS credential broker (running outside your pod), we need a proxy from within the pod (localhost). AWS ECS support in the SDK enforces the use of localhost addresses for fetching credentials.

Using the sidecar proxy has additional benefits. In the steps below to build the image, the default configuration will also cache generated credentials. Misprogrammed user applications (excessive AWS service client generation) will benefit from the caching to avoid performance bottlenecks with repeated calls to STS and AWS Lambda.

Webhook Walkthrough

The webhook is responsible for identifying of pods created those permitted to AWS resources and setting them up with the required secrets and sidecars to call the broker via the proxy. It pulls the functionality in the previous two components together.

This webhook will:

1. Watch for Pod CREATE/DELETE
2. On CREATE:

    a. Identify if the Pod is in a project and service account we know a role can attached to (query to DynamoDB)
    b. Query the OpenShift API server for the desired role on the serviceaccount annotation “eks.amazonaws.com/role-arn”
    c. Register generated auth token in DynamoDB table
    d. Add Secret to namespace containing auth token
    e. Return JSONPatch with:
        i. Add sidecar container for the proxy
        ii. Add environment variables to each container:
            1. AWS_CONTAINER_CREDENTIALS_FULL_URI set to the endpoint of localhost for proxy
            2. AWS_CONTAINER_AUTHORIZATION_TOKEN set to the secret from above
3. On DELETE:
    a. Identify if the Pod previously was given an authorization token
    b. Clean it up if so

Installation

Sidecar Proxy

To leverage the credential broker (running outside your pod), we need a proxy from within the pod (localhost). Detailed below is an effective way to get a sidecar container built inside your cluster honoring the environment variables and secrets we create. The method below sets the cluster up to not only work within this mechanism, it can actively track support/maintenance from Red Hat.

Proxy Sidecar Build

We create a simple container in the default project for this.

Create ImageStream for Container

```
$ oc create is ocp-broker-proxy
imagestream.image.openshift.io/ocp-broker-proxy created
$ oc set image-lookup ocp-broker-proxy
imagestream.image.openshift.io/ocp-broker-proxy image lookup updated
```

Create BuildConfig for ocp-broker-proxy

This proxy makes use of a small and simple nginx.conf identifying the custom endpoint for your credential broker as well as sending an authentication token to the broker (via a Secret). By default, this shared image will be built & deployed any time a change to the Red Hat

ImageStream is detected. To avoid redeploys to your app containers, you can identify the image (not ImageStream) in the broker's webhook configuration later on.

Build_config.yaml:

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
 name: ocp-broker-proxy
 namespace: ocp-iam-broker
spec:
 triggers:
   - type: ImageChange
     imageChange: {}
   - type: ConfigChange
 runPolicy: Serial
 source:
   type: Git
   git:
     uri: 'https://github.com/cuppett/ocp-iam-broker.git'
     ref: master
   contextDir: assets/proxy
 strategy:
   type: Source
   sourceStrategy:
     from:
       kind: ImageStreamTag
       namespace: openshift
       name: 'nginx:latest'
     env:
       - name: OCP_BROKER_LOC
         value: 'https://YOURAPI.execute-api.REGION.amazonaws.com/Prod'
 output:
   to:
     kind: ImageStreamTag
     name: 'ocp-broker-proxy:latest'
```

IAM Broker & Webhook

The broker is the piece which sits as a lambda utilizing its AWS temporary credential. When calling the webhook portion, it will configure incoming pods with a sidecar to callback for their temporary credentials. When the broker is called, it will validate the auth token to sts:AssumeRole to the desired role and serve it to the pod proxy.

Create Cluster Broker Project

To complete the integration, we'll need a few artifacts, secrets and credentials. We'll manage all those from a standard OCP project.

```
$ oc new-project ocp-iam-broker --description "Assigns and gives out temporary
credentials to application projects" --display-name "OCP IAM Broker & Proxy"
```

Create Cluster Broker Service Account

We'll create a standard service account used to query and update:

```
$ oc create sa broker -n ocp-iam-broker
serviceaccount/broker created
```

For the integration, the webhook needs to create and delete secrets to establish auth tokens to the broker service:

```
$ oc create clusterrole work-secrets --verb=create,delete --resource=secret
clusterrole.rbac.authorization.k8s.io/work-secrets created
$ oc adm policy add-cluster-role-to-user work-secrets
system:serviceaccount:ocp-iam-broker:broker
clusterrole.rbac.authorization.k8s.io/work-secrets added:
"system:serviceaccount:ocp-iam-broker:broker"
```

For the integration, the webhook needs to inspect serviceaccounts used by the pod to identify the desired identity (they will also be matched up within the broker):

```
$ oc create clusterrole describe-sas --verb=get --resource=serviceaccount
clusterrole.rbac.authorization.k8s.io/describe-sas created
$ oc adm policy add-cluster-role-to-user describe-sas
system:serviceaccount:ocp-iam-broker:broker
clusterrole.rbac.authorization.k8s.io/describe-sas added:
"system:serviceaccount:ocp-iam-broker:broker"
```

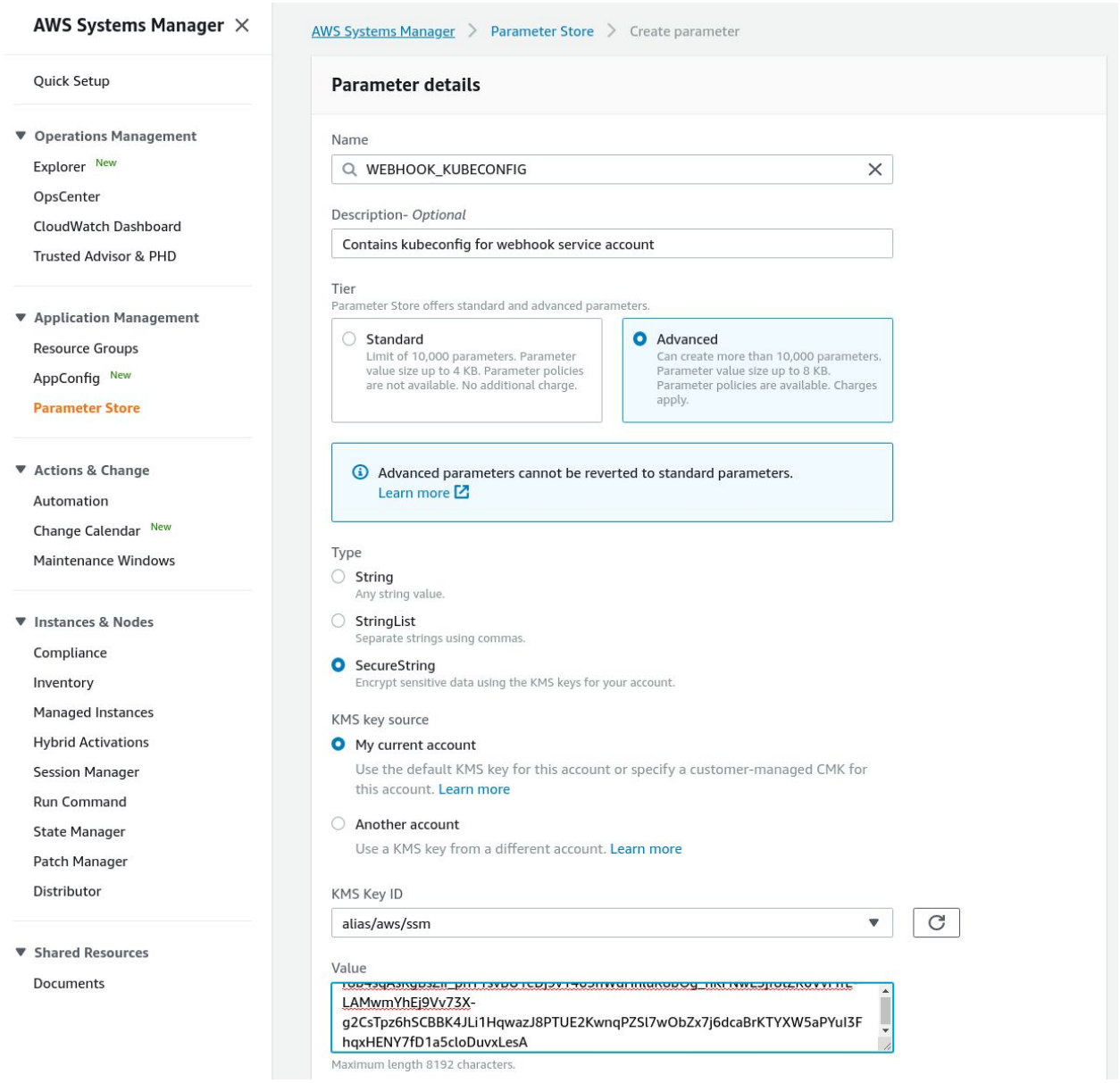
Extract kubeconfig for Webhook Service Account

The kubeconfig is used by the webhook to inspect the service accounts and create authorization secrets. To log in, we need to extract the credential (and save it for the Lambda deployment):

```
$ oc sa create-kubeconfig broker > broker.kubeconfig
```

Create Secure Parameter in Systems Manager

Copy/paste the generated broker.kubeconfig contents into a SSM SecureString parameter. You may use any KMS key (or the default one) you prefer for your account.



Deploy Broker & Webhook via CloudFormation

The broker and webhook are bundled in a lambda function. It can all be registered and configured via CloudFormation.

Creates:

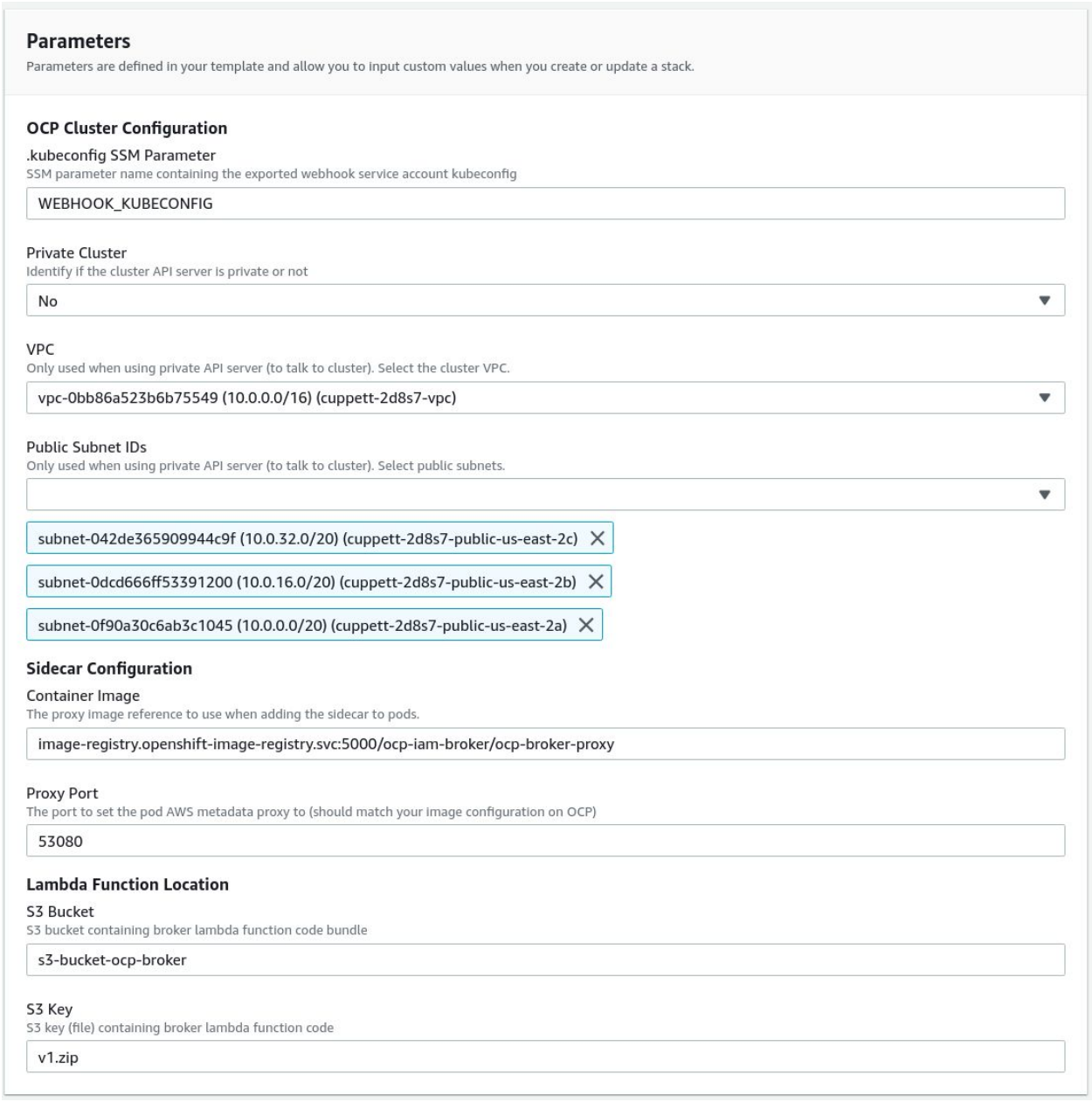
1. DynamoDB table registries (for auth token storage from pods - not the AWS creds)
2. Lambda function (webhook and sts broker) with IAM role
3. API Gateway to invoke webhook and broker from the cluster

Input: S3 location (in your account) of distributed lambda/broker code
Input: kubeconfig SSM parameter
Input: Image to use for the sidecar
Input: Port the proxy image is configured to listen on
Input: Network configuration for the cluster

The current ZIP file and CloudFormation can be found here:

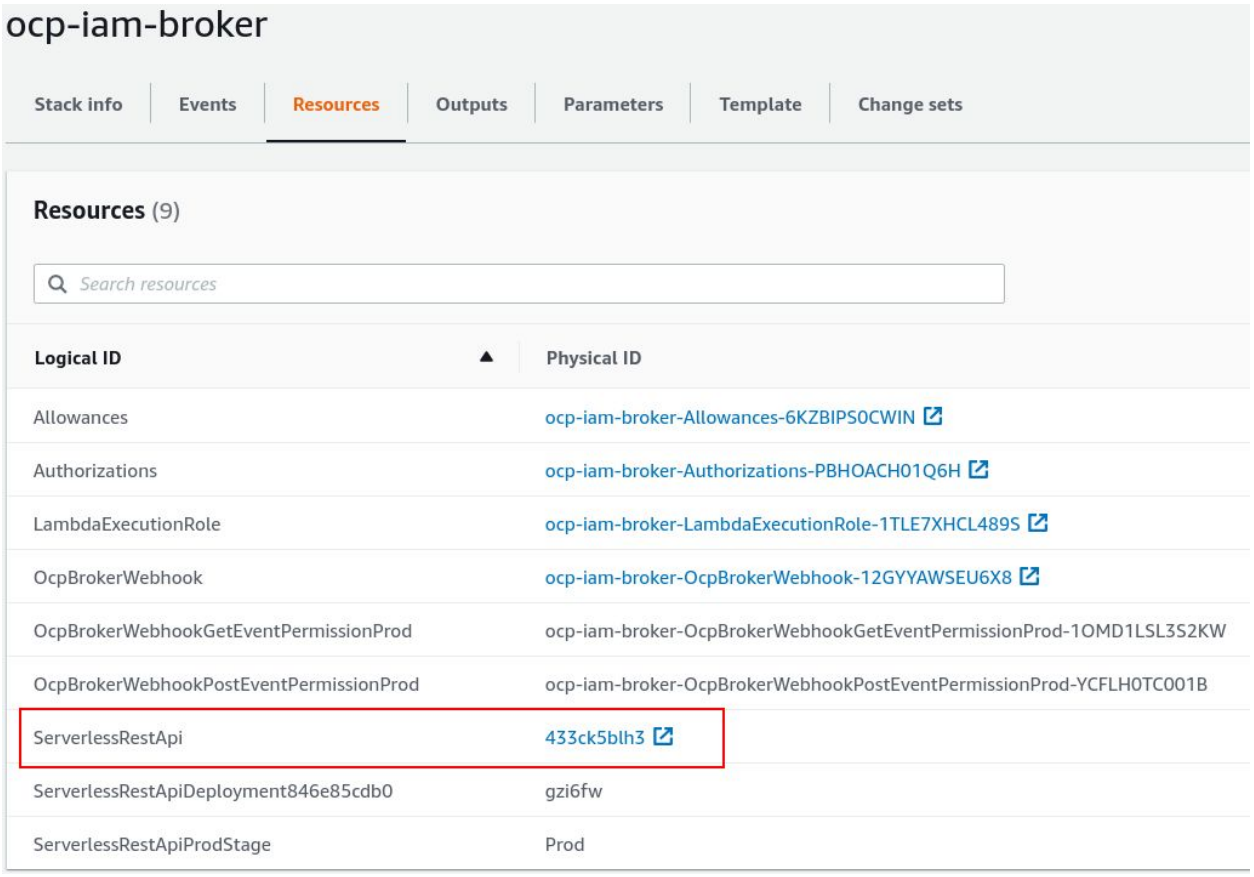
https://github.com/cuppett/ocp-iam-broker/releases

Create Stack



Stack Resources

Once the stack has completed creation, you are able to inspect all the resources created and ensure successful completion. You will need the ServerlessRestApi to register the mutating webhook:



Register Mutating Webhook

/tmp/webhook.yaml:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: ocp-iam-webhook
webhooks:
  - name: YOURAPI.execute-api.REGION.amazonaws.com
    clientConfig:
      url: https://RESTAPI.execute-api.REGION.amazonaws.com/Prod
    rules:
      - operations: [ "CREATE", "DELETE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
```

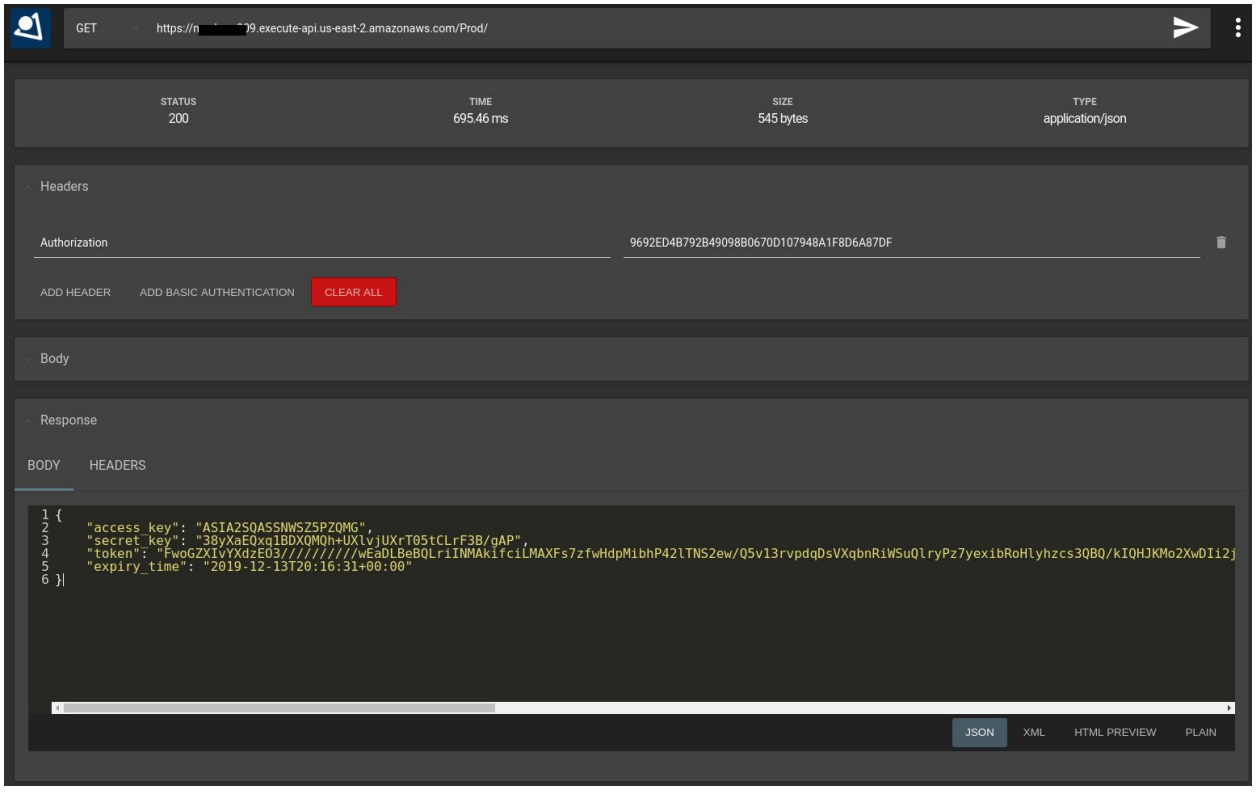
$ oc create -f /tmp/webhook.yaml
mutatingwebhookconfiguration.admissionregistration.k8s.io/ocp-iam-webhook created

Validation

Given there are three, discrete pieces to this solution, it's important they are all functional. Below are various setups which can be used to verify different parts.

Example: Validate Broker Only

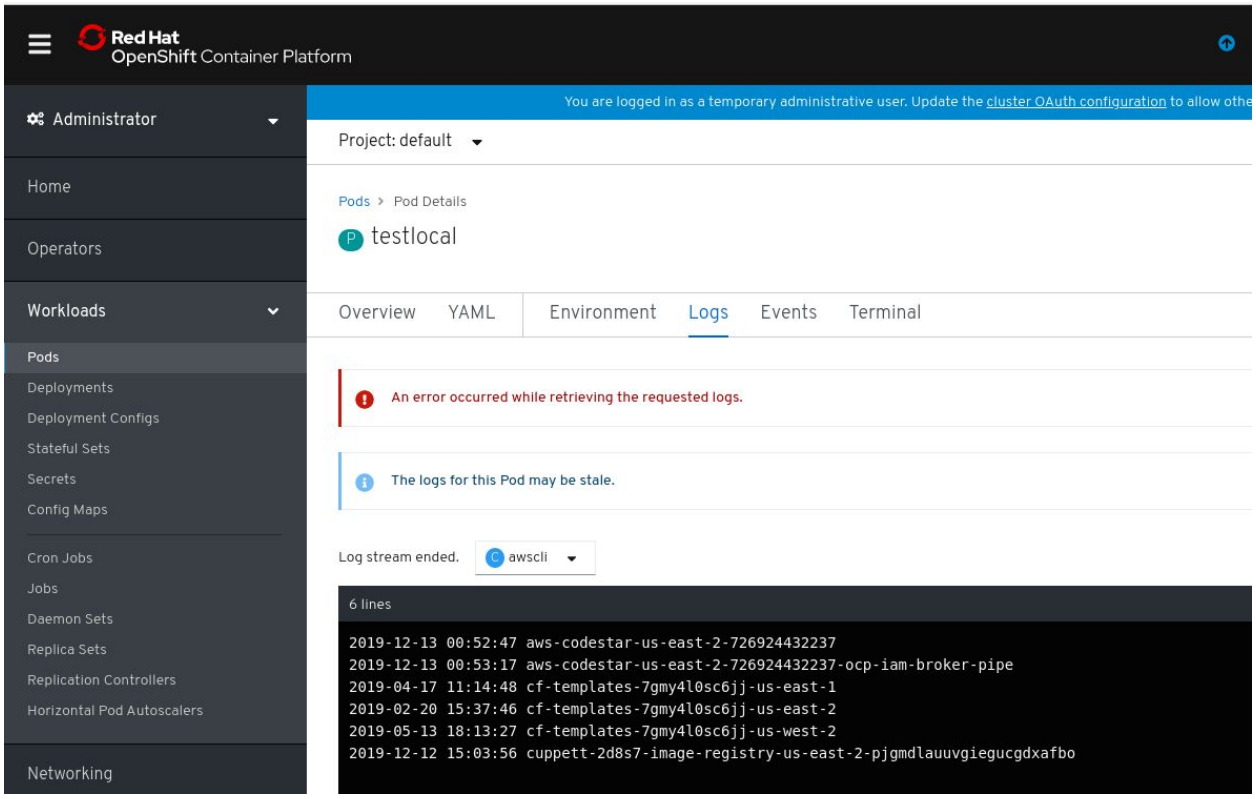
You can validate the broker scenario by creating an authorization row in DynamoDB and running a sample REST call with the Authorization header, ensuring you get a valid credential back:



Example: Validate Proxy & Broker Only

You can validate the proxy<->broker interaction with a simple Pod. Be sure to use a service account and project where the webhook will not interfere. You validate the other two components by inserting into the pod specification the items we'd expect the webhook to inject (your service account still must be able to pull the proxy image and the required authorization row in DynamoDB must be manually inserted):

```
apiVersion: v1
kind: Pod
metadata:
 name: testlocal
 labels:
   app: s3-listing
 namespace: app1
spec:
 containers:
   - name: ocp-iam-broker-proxy
     image:
image-registry.openshift-image-registry.svc:5000/ocp-iam-broker/ocp-broker-prox
y@sha256:fd17b692b955a74bffbf41eeb4e06a9749973efe5658842e2c7bffe64a73259c
   - name: awscli
     image: quay.io/cuppett/aws-cli
     env:
       - name: AWS_CONTAINER_CREDENTIALS_FULL_URI
         value: "http://127.0.0.1:53080/"
       - name: AWS_CONTAINER_AUTHORIZATION_TOKEN
         value: "9692ED4B792xxxx8A1F8D6A87DF"
     command: ["aws"]
     args: ["s3", "ls"]
```



Usage

Using the system as an end user requires 3 main activities:

1. Identifying in DynamoDB valid roles for service accounts & projects to assume
2. Establishing trust from the role to the Broker Lambda role in IAM (for sts:AssumeRole)
3. Configuring the service account in OCP
    a. Annotating with desired, target role
    b. Granting image-puller for the project or service account

Configuring Application Service Account

Create Service Account for Workload

For this solution (and the EKS pod identity solution), service accounts are used to identify and steer workloads to IAM identities.

```
$ oc create sa app-sa -n app1
serviceaccount/app-sa created
```

Annotate Service Account for the Target IAM Role

In DynamoDB, there is a table identifying the roles projects and service accounts can assume. This constrains what is possible/allowed to be served by the broker. The webhook will ensure an entry exists in this registry. The webhook will also check for an annotation on the service account in OpenShift.

```
$ oc annotate sa app-sa
eks.amazonaws.com/role-arn=arn:aws:iam::1111111111:role/s3_reader
serviceaccount/app-sa annotated
```

Allow Pulling the Proxy image

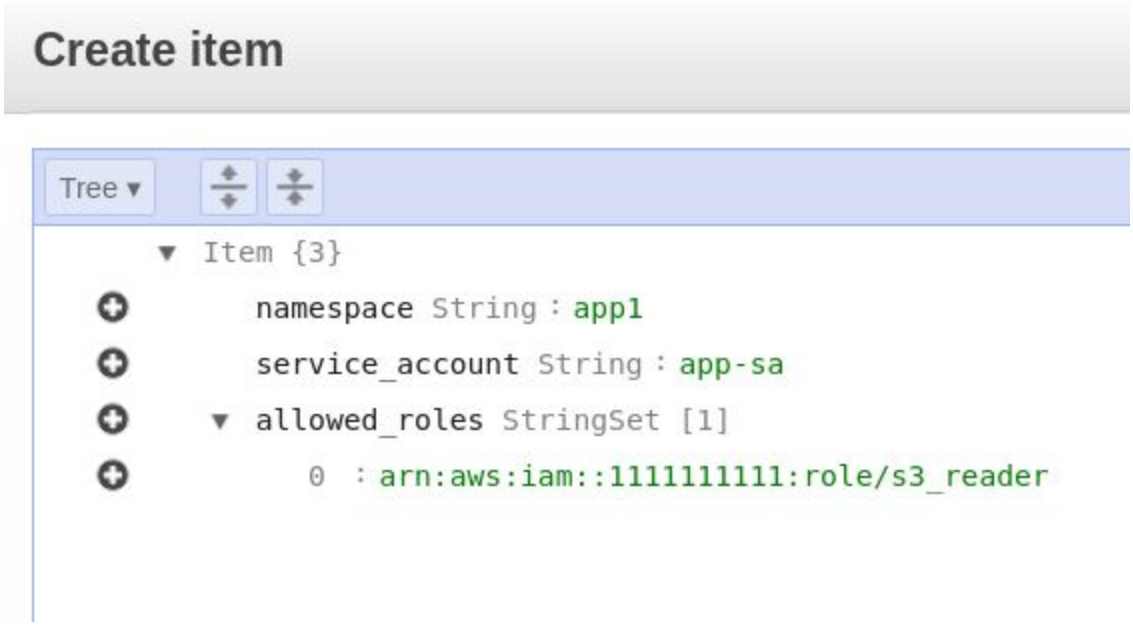
To fetch temporary credentials, a sidecar proxy is required. The image inserted and used is identified to the webhook via configuration directed by CloudFormation. If using the internal registry and separate project (as outlined in this document), you must grant the service accounts which can assume a role permission to pull the proxy:

```
$ oc policy add-role-to-group \
    system:image-puller system:serviceaccounts:app1 \
    --namespace=ocp-iam-broker
clusterrole.rbac.authorization.k8s.io/system:image-puller added:
"system:serviceaccount:app1:app-sa"
```

Configuring the Broker & Webhook

Adding the Target IAM Role to the Service Account (in DynamoDB)

The Allowances table created by the CloudFormation in AWS controls whether this particular combination is allowed. You will insert a new row into the Allowances table similar to below (following our example here):



A particular service account in each namespace may have any number of roles which could be assumed. The annotation on the actual service account in the cluster dictates which one of the allowed ones will be served back by the sidecar.

Submitting a Pod

Once the ServiceAccount is set and the row in DynamoDB is created, you can submit a pod. Following along with the example:

```
apiVersion: v1
kind: Pod
metadata:
  name: s3-listing
  labels:
    app: s3-listing
  namespace: app1
spec:
  serviceAccountName: app-sa
  containers:
    - name: awscli
      image: quay.io/cuppett/aws-cli
      command: ["aws"]
      args: ["s3", "ls"]
```